CS162-Group 35
10/25/2017

Nathan Villegas
Sean Atkins
Trevor Rollins
Edmund Dea

Group 35 Project Design Plan

I. Design
   A. class Tool
      1. Public:
         a) Tool() : strength(2), type(?) { } // Default constructor
         b) Tool(int tempStrength, int tempType) : strength(tempStrength), type(tempType) { } // Overloaded constructor
         c) Get-Set functions
            (1) Void getStrength() const { return strength; }
            (2) Void getType() const { return type; }
            (3) Void SetStrength(int tempType) { type = tempType; }(multiply by 2, but make invisible to user)
            (4) Void setType(char *tempType) { type = tempType; }
      2. Protected:
         a) Int strength
         b) Char *type
   B. class Rock : protected Tool
      1. default constructor that sets the strength to 1. initialize the type field using 'r' for Rock.
      2. non-default constructor which will take in an int used to initialize the strength field. initialize the type field using 'r' for Rock.
      3. Public:
         a) Int fight(Tool)
            (1) Rock's strength is doubled (temporarily) when fighting scissors, but halved (temporarily) when fighting paper.
            (2) The strength field shouldn't change in the function.
            (3) Return temp strength
   C. class Paper : protected Tool
      1. default constructor that sets the strength to 1. initialize the type field using 'p' for Paper
      2. non-default constructor which will take in an int used to initialize the strength field. initialize the type field using 'p' for Paper
      3. Public:
         a) Int fight(Tool)
            (1) Paper’s strength is doubled (temporarily) when fighting Rock, but halved (temporarily) when fighting Scissors.
            (2) The strength field shouldn't change in the function.
            (3) Return temp strength

D. class Scissors : protected Tool
    1. default constructor that sets the strength to 1. initialize the type field using 's' for Scissors
    2. non-default constructor which will take in an int used to initialize the strength field. initialize the type field using 's' for Scissors
    3. Public:
        a) Int fight(Tool)
            (1) Scissor's strength is doubled (temporarily) when fighting Paper, but halved (temporarily) when fighting Rock.
            (2) The strength field shouldn't change in the function.
            (3) Return temp strength

E. class RPSGame
    1. [Outside of class RPSGame but in same header file]
        a) enum win_type { HUMAN_WIN = 0, COMPUTER_WIN, TIES };
    2. Public:
        a) Void play(); // allows a human to play the rock, paper, scissors game against the computer
    3. Private:
        a) Tool* human; // Your RPSGame must have two Tool* for the human and computer tool because you don't know the new tool they'll select with each round
        b) Tool* computer;
        c) Int winCount[3];
            (1) winCount[0] = human_wins;
            (2) winCount[1] = computer_wins;
            (3) winCount[2] = ties;

F. Makefile

G. class Menu
    1. // Just port in someone's implementation of their menu class

H. Input Validation
    1. // Just port in someone's implementation of their input validation class
    2. Please use **Regular Expressions** to cover all types of input validation
        a) To include RegEx header for cross-platform:

```
// Include header
#if (defined(_WIN32) || defined(_WIN64))
   #include <regex>
   using std::regex;
#elif defined(__unix__)
   #include <boost/regex.hpp>
   using boost::regex;
#endif

// Declare regex object
regex objectName(string regexStringPattern);

// Check if objectString matches the regex pattern
regex_match(string *objectString, regex objectName)
```

II. Action Items
   A. Complete Design [everyone] **(Done)**
   B. Complete Reflection document **(AR: Edmund) (Done)**
   C. Complete Testing/makefile/menu **(AR: Sean) (Done)**
   D. Tool/Rock/Paper/Scissor classes **(AR: Nathan) (Done)**
   E. RPSGame class/input validation **(AR: Trevor) (Done)**
   F. Confirm whether to name headers as ".h" or ".hpp" **(AR: Trevor) (Done)**
      1. Use .hpp files for headers

Submit Code to Git: https://github.com/CS162-Group35/GroupProject

Test Plan

Test Case	Input Values	Driver Function	Expected Outcome	Actual Outcome
R,P,S function utilize both upper and lower selections for user input	R, R, P, S, p, s, 9, f, D	Input validation	Only lowercase or uppercase r, p, and s are accepted.	Only lowercase or uppercase r, p, and s are accepted.
User tool strength menu validates to Y/N	C, 8, y, n, Y, N	Input validation	Y, y, N, n are the only acceptable values	Y, y, N, n are the only acceptable values
P, R, S options for tools only accept P, R, S in both lower or uppercase.	10, a, 9 t, A, a	Input Validation	Lower and upper case versions of p, r, s are only values accepted	Lower and upper case versions of p, r, s are only values accepted
Scoring Functionality returns correct values	Gameplays	Game play functionality and all associated classes	Game scoring is managed correctly	Game scoring was managed correctly and displayed to the user
User input tool strength is correctly handled and only accepts an integer value.	A, n, v, s G, 178, 18, 1	Input validation and gameplay functionality	Custom strength is handled correctly and passed onto the game to be used in the strength halving or multiplying function calls	Custom integer strength was the only value accepted and used in the gameplay.
Exit functionality only accepts Y, y, N, n	9, a, N, 13, y, Y, N, n	Input validation	Game exits with a y or Y. Game play continues with an n, N, otherwise incorrect input is caught and the input validation refreshes	Cerrect input only takes Y, y, N, or N. On the correct input, the game either repeats or exits correctly.
Makefile compiles correctly	C++ make	Files and makefile	Correct object compilation and game creation	Initially no. The make file was re-edited so that correct compile testing would occur.

Reflection

Our collaboratively came up with the design for the group project during our first online meeting last week, which lasted 1.5-hours. We all read through the assignment specs together and each person contributed ideas toward the requirements for each class. For each class, we decided what member variables should be included and then we determined the member functions. At the end of the meeting, we split up the work and each person volunteered to complete one action item, as listed above. All members tracked their work online using Git, and the Git history was used to create this reflection.

Fortunately, the initial implementation of the classes did not deviate from the design specs and the Tic Tac Toe program compiled the first time we all put it together. Although, there were several small warnings that we had to debug during our second meeting, which were exposed when we compiled using Visual Studio, g++ -Wall, and Valgrind.

One minor design change was to seed the random number generator within the main function rather than within the RPSGame class. Trevor determined that it's only necessary to seed the random number generator once and the program could potentially call srand multiple times if multiple instances of RPSGame were created.

A second minor was the addition of comments in each function. Early versions of our code base did not contain many comments, so we added comments to explain each block of code.

A third minor misunderstanding in the design was related to cleaning up of memory. Edmund initially misunderstood why there was an ~RPSGame destructor and a deleteTools function, which deleted all the Tool pointers. Therefore, Edmund tried adding the deleteTools function in the ~RPSGame destructor. However, Trevor explained that this can result in a double deletion of memory since deleteTools is explicitly called in the play function. so Edmund resolved the issue by removing the deleteTools call in the ~RPSGame destructor. This issue taught us that it is very important to

Lastly, the initial design of the program did not include a virtual destructor for the Tool class and the derived classes. Once we tried compiling the program on Flip with warnings enabled, we received a compiler warning saying "deleting object of polymorphic class type 'Tool' which has non-virtual destructor might cause undefined behaviour". This problem taught us that when we're using polymorphism and delete an object from a derived class, the base class is required to have a virtual destructor. If we don't have a virtual destructor in the base class and delete an object in a derived class, then only the base class destructor will be called, which can cause unexpected bugs in the program. This issue was resolved once we added a virtual destructor in the Tool class as "virtual ~Tool() {}".

When we built the program in Visual Studio, we found one compiler warning reporting "conversion from 'double' to 'int', possible loss of data". This issue occurred because each of the Rock, Paper, Scissors classes contained a fight function that returned the "opponent's tool strength * 0.5". Since the opponent's tool strength is an int and we're multiplying by a double, 0.5, there is implicit conversion of the tool strength int to a double. Implicit conversion causes the product type to change from int to double. The problem is that the fight function's return type is int. Therefore, implicit conversion occurs a second time to change the product from double to int, which is converting from a decimal to floating-point number. Since double is a floating-point integer type that typically contains twice as many bits in memory than int types, there could possibly be a loss of information when converting from double to int. This issue was resolved by changing the formula from

“opponent’s tool strength * 0.5” to “opponent’s tool strength / 2”. This works because dividing an integer by an integer removes any implicit conversion of number types. This lesson taught us that it’s important to always check code where implicit conversion can cause loss of information.